Mixed-Tool Performance Analysis on Hybrid Multicore Architectures

Peng Du*, Piotr Luszczek*, Stanimire Tomov*, Jack Dongarra*†‡
*University of Tennessee Innovative Computing Laboratory
†Oak Ridge National Laboratory
‡University of Manchester

***Abstract*—This paper proposes a triangular solve algorithm with variable block size for graphics processing unit (GPU). By using diagonal blocks inversion with recursion, this algorithm works with tunable block size to achieve the best performance. Various methods are shown on how to make use of existing profiling tools to successfully measure and analyze performance of this algorithm. We use some of the most popular CPU and GPU profiling tools for their advantages and overcome their disadvantages with several new techniques to analyze the performance and relationship of different components of applications. With the presented methodologies, insight information is produced which helps to understand and tune the proposed algorithm and considerably improve the performance of the solver itself as well as the application using it.**

I. Introduction

CUDA visual profiler [17] is by far the most popular tool for GPU code profiling. It allows users to gather information about kernel execution and memory transfer operations. The profiler can be used to identify performance bottlenecks or to quantify the benefit of optimizing a single kernel, but it lacks the ability to analyze the interaction between different kernels, and unfortunately this is exactly what is needed to tune the triangular solver – a code that has several GPU kernels running together. Our goal is to minimize the total run time rather than that of each kernel – in some cases, reducing the run time of a particular kernel can adversely impact the run time of other kernels, resulting in larger overall execution time. The algorithm we are proposing consists of three kind of kernels. Different kernels have different proportion in the total execution time, and the time distribution varies when parameters of the algorithm are adjusted. To obtain more insight of the execution to tune the triangular solver it was necessary to interface with TAU, a popular CPU profiler tool, which provides insight into the impact of one kernel on others.

The rest of the paper is organized as the following. Section II describes the application background of the paper and introduces the triangular solve algorithm with variable block size that is to be analyzed and tuned throughout the text. Section III discusses the disadvantages of low level profiler, namely the CUDA visual profiler, and section IV shows the disadvantage of the high level profiling tools with GPU program. Section V explains how to interface TAU to profile GPU program and section VI and VII discuss usage of TAU to optimize the proposed algorithm. Section VIII shows the experiment result and IX gives a brief of the related work.

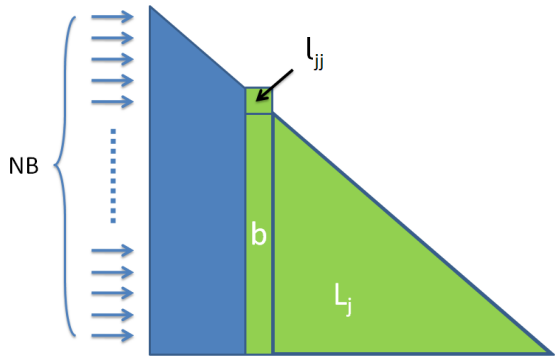

Fig. 1. Single step of diagonal element inversion in the block inversion kernel.

II. Application Background

A. Theoretical Setting

The triangular solver is a part of general linear equation solver: given a triangular matrix T of dimension n by n ($T \in \mathbb{R}^{n \times n}$) and a right hand side matrix B of dimension n by $nrhs$ ($B \in \mathbb{R}^{n \times nrhs}$), the solution matrix X is governed by equation

$$TX = B. \tag{1}$$

If $nrhs > 1$ then this is a Level 3 BLAS operation [9], [8], [6], [7]. It has sixteen variants depending on whether T is upper or lower triangular, has a unit or non-unit diagonal, is multiplied from left or right in a transposed or non-transposed form. Very similar analysis can be applied to any of the variants so for conciseness we only discuss the L-L-N-N variant: multiplication from left, lower triangular T, non-transpose and non-unit diagonal. The letters correspond to the first four parameters to BLAS routine xTRSM (x may be substituted for 'S' or 'D' – for single or double precision, respectively). For clarity we will use L rather than T to represent lower triangular matrix or submatrix.

B. Block Algorithm for Diagonal Inversion

A standard practice of high performance BLAS operation is to use a variant of a block algorithm [3], [13]. In our case

we partition the triangular system of equations:

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$
$$\Rightarrow \begin{cases} x_1 = L_{11}^{-1} b_1 \\ x_2 = L_{22}^{-1}(b_2 - L_{21} x_1) \end{cases} \quad (2)$$

The algorithm has three main computational components:

1) The inversion of the diagonal blocks of submatrices L_{11} and L_{22} (the inversion becomes simply a reciprocal of diagonal elements when block size $NB = 1$.)
2) Matrix multiplication (GEMM) to produce the solution to one block of x.
3) GEMM to update the remaining blocks of b.

BLAS uses forward substitution algorithm which is equivalent to $NB = 1$ of algorithm (2), and the performance of BLAS TRSM suffers from this choice because the inversion of the diagonal blocks, L_{11}^{-1} and L_{22}^{-1}, is on the critical path of the execution. This means all the diagonal block inversion or element reciprocal has to be done in a sequential fashion, even though there is no actual dependencies between them. Also, since the diagonal blocks inversion is much slower than the other two parts (the GEMMs), this algorithm is not suitable for direct use on GPU.

We use a different approach to ensure high performance by choosing an algorithmic variant with higher level of available parallelism. The bottleneck of the serialized diagonal block inversion is removed by spawning a set of thread blocks that perform all the inversions in parallel as there is no dependence between each inversion. The time of part 1 of the TRSM algorithm is essentially reduced to the time to perform inversion on just one block. Since the diagonal block inversion also determines the numerical stability for the whole triangular system solver, a high performance triangular inversion algorithm with good stability is be preferred. Out of the two candidates in [10], the following one is chosen:

For $X = L^{-1}$, problem size $n \times n$, x_{jj} and l_{jj} being the j^{th} elements on the diagonal of X and L, respectively.

$$\begin{aligned} &for \quad j = n:-1:1 \\ &\quad x_{jj} = l_{jj}^{-1} \\ &\quad X(j+1:n, j) = X(j+1:n, j+1:n) L(j+1:n, j) \\ &\quad X(j+1:n, j) = -x_{jj} X(j+1:n, j) \\ &end \end{aligned} \quad (3)$$

This method makes n calls to TRMV (triangular matrix-vector multiplication), and on a GPU this is implemented as GEMV (regular matrix-vector multiplication) with zeros enforced on upper triangular of the matrix to avoid conditional branches that separate the execution path between upper and lower triangular parts. It has been proven that the stability of this method is as good as using forward substitution for inversion[10] , so it is used for our diagonal block inversion kernel.

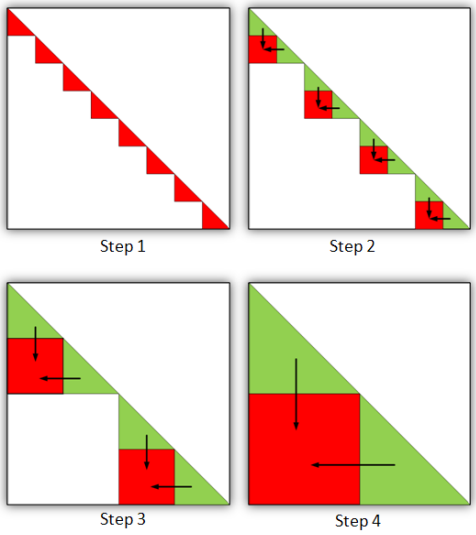

Fig. 2. Recursive algorithm for inversion of increasingly larger diagonal blocks.

Figure 1 shows the inversion kernel mid-way through the execution at step j. A 1D thread block of size NB is used to invert this matrix. Note that b has a single column of elements. The green blocks are those involved in this step of operation. As j decreases, the green lower triangular matrix becomes larger and eventually it will cover the whole matrix except for the first column. In other words, at step j, the triangular matrix used in step $j-1$ is still used, which requires matrix used in step $j-1$ to stay in shared memory. As j reaches 0 (the first column), the whole matrix would stay in shared memory. Nowadays, most of GPUs have 16KB shared memory. For a non-blocking triangular inversion on a GPU, the largest possible dimension size that is a multiple of 16 (half a warp of threads on a GPU) is 32. Using the CUDA Profiler, we determine the total time of inverting a 32 by 32 triangular block and the time to invert the two 16 by 16 blocks on the diagonal followed by Triple-Matrix update (TM update) to finish the block at the bottom left corner. Such an algorithmic variant is preferable to using algorithm from Equation (3) on the whole 32 by 32 block.

C. Variable Block Size Algorithm

While the diagonal block inversion algorithm is limited to at most 32×32, inversion of larger blocks can be done in a recursive manner [18] with the inner block size (non-blocking triangular inversion) being 16.

Figure 2 is an example of how the recursive inversion algorithm works. At step 1, the non-blocking inversion algorithm is applied to all the diagonal blocks of size 16×16, denoted by red blocks. Then in the following three steps, we applied GEMM update to the corner blocks in red with two matrix multiplications that involve three matrices. This is described

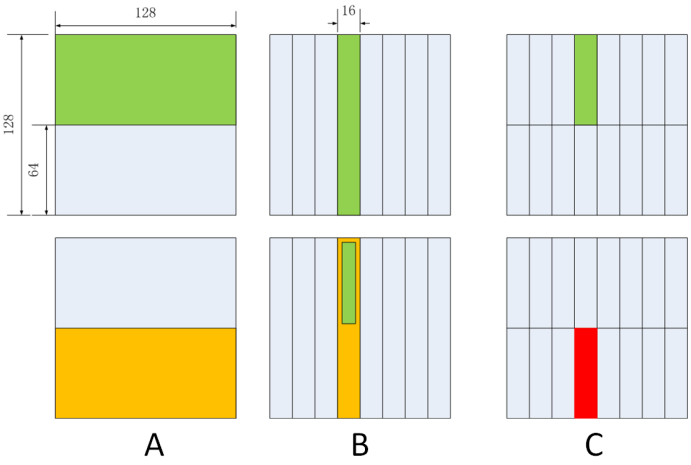

Fig. 3. Illustration of a race condition issue when mixing input and output parameters for GEMM.

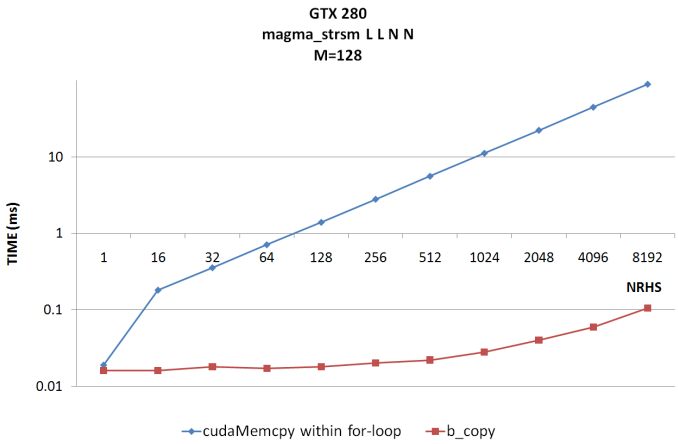

Fig. 4. Time to perform a copy of a matrix of a given size using multiple calls to cudaMemcopy and a custom b_copy routine optimized for the task.

in (4):

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} X_{11} & 0 \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}$$

$$\Rightarrow \begin{cases} X_{11} = L_{11}^{-1} \\ X_{22} = L_{22}^{-1} \\ X_{21} = -L_{22}^{-1} L_{21} L_{11}^{-1} \end{cases} \tag{4}$$

In each step of TM update, a triangular block that has the red block X_{21} to update is called a 'page'. In figure 2 for example, step 2 has four pages, and step 4 has one page. The number of pages is the number of TM updates in a certain step. On a GPU, all pages in any step are updated simultaneously because there is no dependency between them. Volkov's GEMM algorithm [21] is modified for this purpose. At each step, two kernels are launched one after another to do $L_{21} \leftarrow L_{21} L_{11}^{-1}$ and $L_{21} \leftarrow -L_{22}^{-1} L_{21}$ respectively. These two kernels are named triple_sgemm_update_64_part1_L, and triple_sgemm_update_64_part2_L, '64' respresents the problem size of the current GEMM, which can vary. And these two kernels update the inversion of a size twice the number in the routine name. For example, triple_sgemm_update_64_part1_L, and triple_sgemm_update_64_part2_L finishes the inversion of blocks of size 128.

This dual-kernel setup as opposed to fusing two GEMMs into one kernel is to avoid race condition; because L_{21} is the output of the first GEMM and the input of the second GEMM, a Read-After-Write (RAW) hazard.

D. Implementation Issues

Volkov's GEMM uses 64x16 block size, of which there are $(M/64) \times (N/16)$ thread block in a 2D grid. When there is more than one row of thread blocks in the grid, an in-place hazard exists for GEMM ($C = AB + C$) when B is the same buffer as C. This affects part of the TM update, as well as CublasSgemm, which is also based on Volkov's GEMM.

Figure 3 illustrates this issue when the grid has two rows and multiple column blocks of threads performing GEMM. In such a scenario, one thread block $T(1,4)$ touches all of the green section of the data in matrix A and B and writes result to the green part of C. A thread block $T(2,4)$ on the second row and the same column in the grid is about to do the same thing but only slightly later than $T(1,4)$. Since B and C are stored in the same buffer, T(2,4) uses B that has already been overwritten by $T(1,4)$. Thus, it results in a race condition. This happens both in our TM update and in the second part of our triangular solver algorithm, $x_1 = L_{11}^{-1} b_1$. The solution for the TM update is to use an intermediate buffer centrally symmetric to C (L_{12} in (4) in the current page) so that B and C are on longer same in GEMM. The result is copied to L_{21} after all thread blocks finish, and L_{12} is set to 0.

The solution for the GEMM that produces the triangular solve for result x is discussed in Section III.

III. Low-Level Profiling with CUDA Profiler

Due to the race condition of in-place update when performing $b_i = L_{ii}^{-1} b_i$, a separate buffer x is created on the GPU so that this hazard is removed by doing $x_i = L_{ii}^{-1} b_i$. After x is filled with the whole solution, the content is transferred to buffer b.

With the presence of LDB(leading dimension of buffer b), it is not uncommon for the right hand side to be a block of a larger matrix. The stride, then, between the starting address of each column of b is LDB which could be larger than the length of the column M. Clearly, cudaMemcpy() can only be used when $LDB = M$, otherwise the copy has to be done column by column. The initial implementation of this memory copy was done with successive calls to cudaMemcpy() within a for-loop. Even though coalesced access to the GPU device memory [15] is guaranteed in each step of the loop, profiling shows the time of this memory access has become the dominant factor of the whole TRSM.

To best utilize the bandwidth of device-to-device communication and to fully take advantage of the parallelism

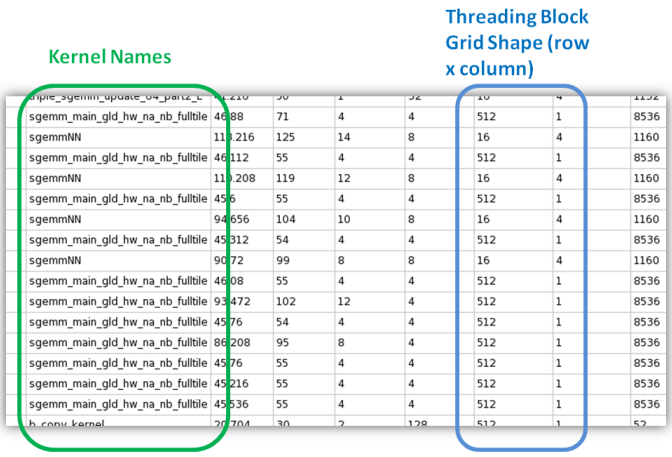

Fig. 5. Sample graphical output from the CUDA Profiler running the cublasSgemm() routine.

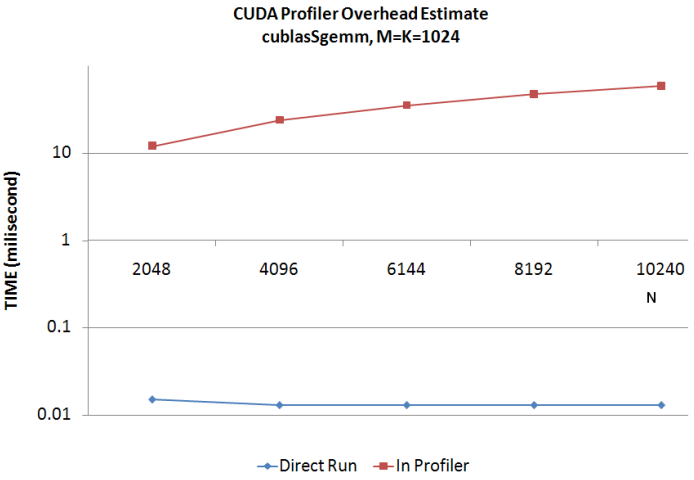

Fig. 6. Timing of cublasSgemm() with and without the CUDA profiler.

provided by its large amount of cores, we optimize the memory copy by launching GPU kernels with the 2D grid of N column of threading blocks. Each thread block has a 512 by 1 configuration. For each column of b, a total number of $\lceil M/512 \rceil$ thread blocks pull the data from their corresponding column to the same column in b, and all data in x is copied to b simultaneously. The routine that invokes the memory copy kernel is called b_copy().

Figure 4 shows the comparison of these two implementations. The time of memory copy is decreased from the initial version by a factor of around 1000.

IV. Limitations of High-Level Profiling with CUDA Profiler

Out of all the three main components of the triangular solver, cublasSgemm() is a natural candidate for optimization: it is an order $O(n^3)$ operation which makes its run time grow quickly especially with multiple right hands and when problem size is large. It is therefore crucial to tune its performance to maximize overall performance of the triangular solver.

Internally, cublasSgemm() calls different CUDA kernels to deal with various calling sequences of GEMM (keep in mind that GEMM takes 13 parameters as an input). And even for the same calling sequences cublasSgemm() calls different kernels depending on the shape of the input matrices. Figure 5 shows a sample output of the Linux CUDA profiler for the triangular solver run with M=1024 and N=128. The names in the green rectangle are the actual computational kernels that are called inside cublasSgemm(); their corresponding thread block grid shape is shown in the blue rectangle. Notice that when dealing with a CUBLAS routine such as cublasSgemm(), the profiler in its current form has limitations that stand in the way of a more informed performance analysis. Dealing with multiple kernels in a single run is one such limitation: cublasSgemm() is used to perform both $x_1 = L_{11}^{-1} b_1$ and $b_2 = (b_2 - L_{21} b_1)$ operations. However, it is not clear which kernels in the green rectangle are called from which instance of cublasSgemm().

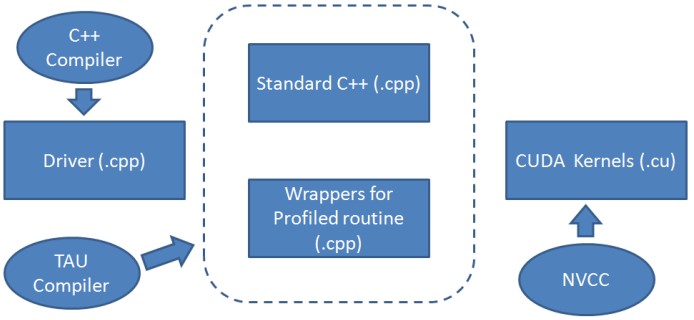

Fig. 7. Program structure to instrument CUDA kernels with TAU.

Another limitation of the CUDA Profiler comes from its use of hardware performance counters. The fact that counters are shared between threads, warps, and GPU multiprocessors practically eliminates the possibility for reliable absolute counter readings. Instead, the use of relative performance trends is advised [11].

Finally, to establish the practical usefulness of the profiler for our purposes, we timed one of the most important CUBLAS routines: cublasSgemm(). Figure 6 shows the obtained timings in two scenarios: with and without the CUDA profiler attached. The figure clearly indicates high overhead associated with running the compiler , and this confirms the importance of relative rather than absolute profiling mentioned earlier [11].

```
extern "C" void
diag_strtri_W(int M, char uplo, char diag, float *A,
              float *d_) {
  diag_strtri (M, uplo, diag, A, d_dinvA, lda);
  cudaThreadSynchronize();
}
```

Fig. 8. An example of a wrapper for routine diagi_strtri().

```
extern "C" void
diag_strtri_W(int M, char uplo, char diag, float *A,
              float *d_dinvA, int lda) {
  TAU_PROFILE(
  "void diag_strtri_(int, char, char, float *, float *,
int) C [{magma_strsm_main.cpp} {27,1}-{31,1}]",
  " ", TAU_USER);

  diag_strtri (M, uplo, diag, A, d_dinvA, lda);
  cudaThreadSynchronize();
}
```

Fig. 9. TAU instrumentation for the wrapped CUDA kernel call.

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        7,394        7,433           1           1    7433783 int main(int, char **)
0.5          0.823           39           1          21      39711 void magmablas_strsm(
0.4             27           27           1           0      27561 void diag_strtri_(int,
0.1              8            8           9           0        910 void cublasSgemm_2_(
0.0              3            3          10           0        312 void cublasSgemm_1_(
0.0           0.02         0.02           1           0         20 void b_copy__(int,
```

Fig. 10. Text output from TAU after running triangular solve routine.

V. Interfacing CUDA with TAU

TAU (Tuning and Analysis Utilities) [19] Performance System® is a portable profiling and tracing toolkit for performance analysis of parallel programs written in various languages such as Fortran, C, C++, Java, and Python. The versatile array of performance analysis methods available in TAU compelled us to use it for understanding the behavior of our CUDA code.

The first issue to overcome is establishing the interface between the CUDA compiler tool-chain and TAU's compiling framework. TAU allows specifying a C++ compiler with the `-c++` option. The list of supported compilers is very comprehensive and allows for use of products from many vendors on a wide range of platforms. In contrast, the CUDA environment is specific to a single vendor that works only with one brand of GPU accelerator. The current (as of this writing) release of TAU 2.19.1 does not support NVIDIA's CUDA technology.

We are interested in profiling a CUDA routine called diag_strtri() that performs diagonal block inversion. The routine contains the standard CUDA syntax: the triple angle bracket notation of the form `<<<...>>>`. Such code has to be compiled by the CUDA compiler called NVCC. As mentioned above, NVCC is not on the C++ compiler list of TAU, and NVCC treats files with `.cpp` extension as standard C++ file. This means a simple name extension change from `.cu` to `.cpp` will break the compilation of the CUDA code.

To solve this issue, all the non-kernel related routines are brought out into a separate file, resulting in a program structure is shown in figure 7. Upper level C++ routines reside in the C++ file. CUDA kernels stay in the `.cu` file(s), and routines like diag_strtri that need to be profiled but have CUDA kernel syntax will be redirected to a wrapper routine in a C++ macro. As an alternative to binary instrumentation, all C++ routines whose source code cannot be accessed, are wrapped. This facilitates TAU's automatic instrumentation at the source code level.

Figure 8 shows how a call to a CUDA kernel is wrapped to make the CUDA-specific syntax invisible to the TAU's C++ compiler. Figure 9 shows the instrumentation code inserted automatically by TAU.

Once TAU is properly set up as described above, the default GNU compiler is replaced in the Makefile with the `tau_cxx.sh` command that produces an instrumented executable.

A number of options are provided to view the result of the execution, for example, the text profiling output for timing information and Jumpshot [24] for viewing the trace files. The configuration of these two routines are straightforward [5].

A typical text profiling result is shown in figure 10, and a tracing image example is presented in figure 11.

For our triangular solver, four routines are profiled:

- The diagonal inverse routine diag_strtri(), labeled 'diag_strtri'
- The matrix multiplication that updates the solution, CublasSgemm(), labeled 'sgemm1'
- The matrix multiplication that update the right-hand side, CublasSgemm(), labeled 'sgemm2'
- The memcopy from temporary buffer to the solution buffer, b_copy(), labeled 'b_copy'

So far we have described a syntactic integration between TAU and the CUDA environment. A more nuanced interaction occurs for synchronization in the profiled code fragments.

CUDA's kernel launch and some CUDA utility routines are non-blocking. They return immediately and the execution on CPU continues. This gives CUDA the ability to schedule and run more than one kernel in parallel and overlap communication with computation. Attention has to be paid when timing CUDA kernels and utility routines because some of them, such as cudaMemcpy() for example, do have a synchronization call to cudaThreadSynchronize() at the end. Others, such as CublasSgemm(), do not. Profiling tools require a clear delineation of the start and end of the routine being analyzed. Without any further changes to profiling wrappers, returning from a kernel launch marks the wrong finish time because the kernel might still be running on the GPU. As an example, the profiling code is run with and without a call to cudaThreadSynchronize(). The results are shown in figure 11. Without a call to cudaThreadSynchronize() at the end, the first three kernels seem to take a very short time while the very last step of the execution, which is b_copy() that copies data to the result buffer, seems to take a few orders of magnitude longer. In a homogeneous multicore environment, such a large difference would clearly indicate an error in the profiling layer but wide performance variability of an hybrid multicore system is very likely [23]. This is mostly due to the control logic constraints of the GPU hardware and how they get magnified by high levels of parallelism – a detailed explanation of such variation is beyond the scope of this writing.

As a consequence of lack of synchronization, one might be led to the wrong conclusion that it is not necessary to tune the first three kernels because, as evidenced by the traces,

Fig. 11. Jumpshot window with TAU trace without explicitly calling cudaThreadSynchronize() inside profiled CUDA routines.

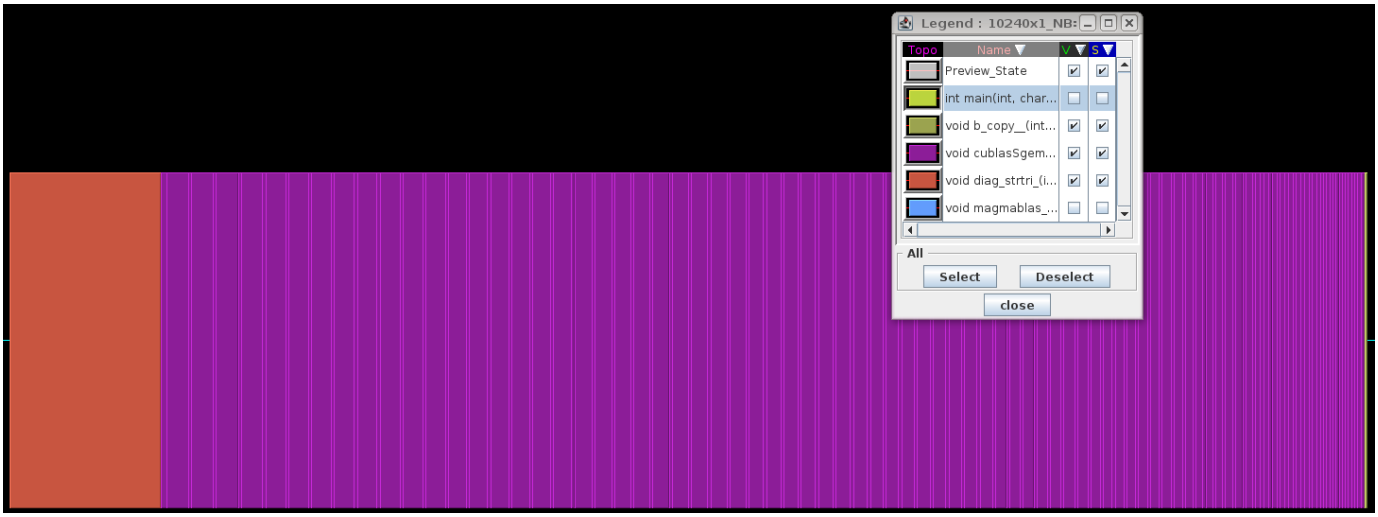

Fig. 12. Jumpshot window with TAU trace with explicitly calling cudaThreadSynchronize() inside profiled CUDA routines.

the performance is dominated by the speed of the device-to-device memory copy operation. In reality, however, TAU captures only the time to launch a kernel for the first three routines. The actual time to run these routines is credited to b_copy() because it finishes with a call to cudaThreadSynchronize(). The correct time distribution is shown in figure 12: cudaThreadSynchronize() call was added after invocation of kernels that don't synchronize the threads. The figure clearly shows that cublasSgemm() consumes the most amount of time.

VI. Profiling CUDA with TAU

With all the tools and wrapping correctly set up, the insight of the TRSM algorithm can be obtained. Figure 13 shows the Gflop/s performance of STRSM with respect to different block sizes. The figure indicates that the best choice apparently is at NB=128. To better understand why this is the case, we turn to TAU to show the distribution of time among the routines invoked by the triangular solver. Figure 14 shows the time distribution of diag_strtri() and cublasSgemm() and memory copy.

diag_strtri() has a slow growth of time when NB$\leq$ 128, and then the growth rate seems double. cublasSgemm time, on the other hand, has a quick drop as NB goes from 16 to 32, then it slowly decreases all the way from NB=16 to NB=512.

The sum of these two has its lowest point at $NB = 128$, which backs up the Gflop/s result well. At $NB = 256$ and above the sum time starts to increase. Even though CublasSgemm still keeps taking shorter time, the increase in diag_strtri() has compromised the decreasing trend.

diag_strtri() is the critical factor in this algorithm. If from $NB = 128$ and above the growth of diag_strtri() time could be a little bit slower, $NB = 256$ would have become the lowest point on the sum time line in 13, which would in turn translate to an even higher Gflop/s.

We want to find the causes of this exponential growth and if there is any ingredient in this recipe that can be improved.

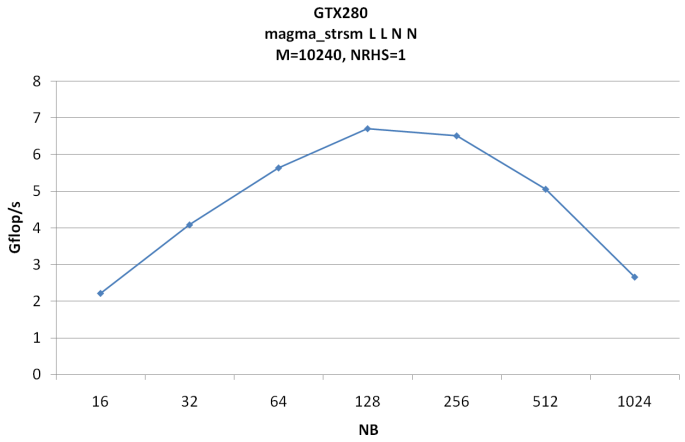

Fig. 13. Performance of the triangular solver with varying blocking factors NB.

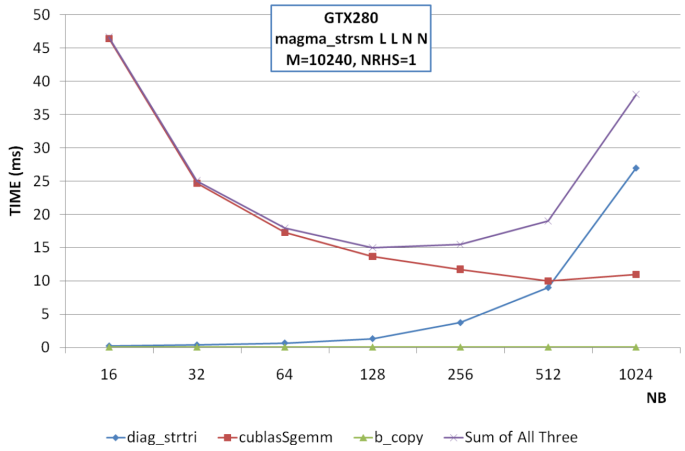

Fig. 14. Distribution of time spent in main time-consuming routines of the triangular solver for varying blocking factor NB.

VII. Decompositional Performance Analysis through Profiling

There are several possible reasons for the rapid increase of run time of diag_strtri():

1) TM update is $O(n^3)$
2) The communication overhead for TM update is $O(n^2)$

	triple_sgemm_update_64_part2_L	196.8	208
	triple_sgemm_update_above64_part1_L	558.592	567
	triple_sgemm_update_above64_part2_L	647.552	658
	triple_sgemm_update_above64_part3_L	149.344	158
	triple_sgemm_update_above64_part1_L	2059.23	207(
	triple_sgemm_update_above64_part2_L	2567.14	2575
	triple_sgemm_update_above64_part3_L	279.264	290
	sgemmNN	320.704	329

Fig. 15. The relevant portion of the CUDA Profiler output after running cublasSgemm inside the triangular solver.

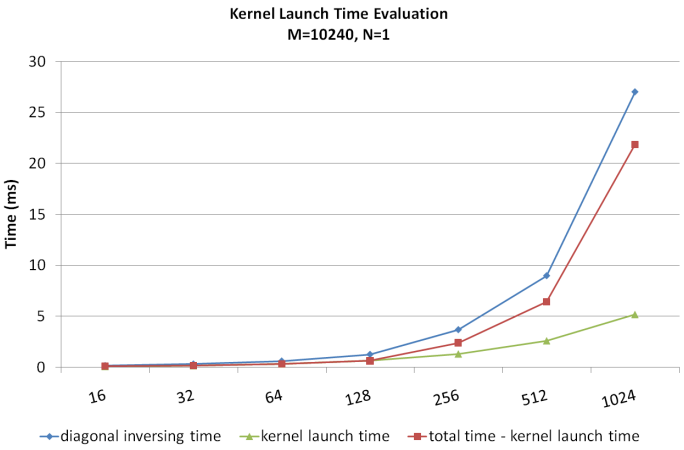

Fig. 16. Comparison of thread launch time and computation time for various blocking factors.

3) The number of thread blocks in each step of the TM update increases, challenging the scheduling capability of CUDA
4) Extra memory copies used to deal with the in-place matrix multiplication issue in TM update when $NB > 128$
5) Kernel launch overhead increases as NB grows

(1) and (2) are because that GEMM in triangular matrix inversion is $O(n^3)$ complexity [4].

When the number of thread blocks increases, the CUDA scheduler might not be able to keep up with the increased workload and the overhead might be an issue. To prove or disprove this conjecture, we port the code to Fermi (C2050) which has 448 cores – almost double that of GTX280. Also, Fermi has a dual-warp scheduler [16]. With more computational resources and a broader execution channel, the problem caused by too many thread blocks could be at least suppressed. Unfortunately, by comparing the output of CUDA profiler on both GPUs, we see no such effect. The run time of the triple-matrix update still increases when $NB > 256$. This disproves our initial conjecture: CUDA seems to be able to handle large workload consistently well.

Possibility 4 comes from the observation that the time of diag_strtri() only starts to increase when $NB >= 256$, and this is exactly the case when extra copy is used to prevent the in-place matrix-matrix multiplication issue in TM update. The kernel that does the extra copy is named triple_sgemm_update_above64_part3_L.

Figure 15 is the output of CUDA profiler for a run with $M = 1024, N = 128, NB = 512$. $NB = 512$ is selected so that we could see the profiling of two consecutive triple_sgemm_update_above64_part3_L, and they are the selected two rows in figure 15. The first row belongs to the TM update that updates an inversion of dimension 256, and the second is 512. We see times quadruple from 256 to 512 with part 1 and part 2 of TM update, while time of triple_sgemm_update_above64_part3_L barely doubles. Clearly this extra copy is not contributing to the exponential increase issue of diag_strtri().

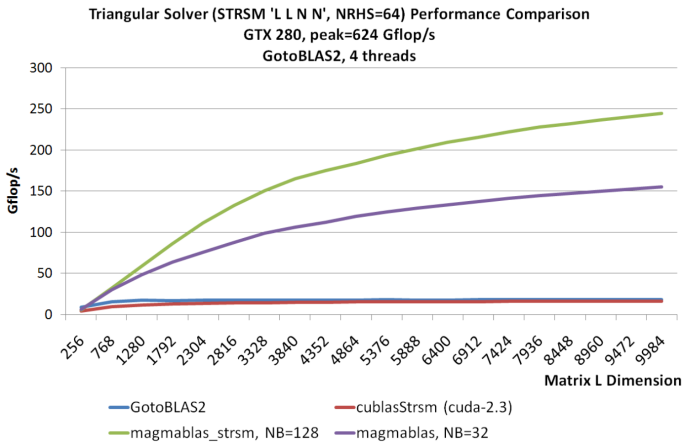

Fig. 17. Triangular solver performance on various hardware with different software implementations When $M > N$.

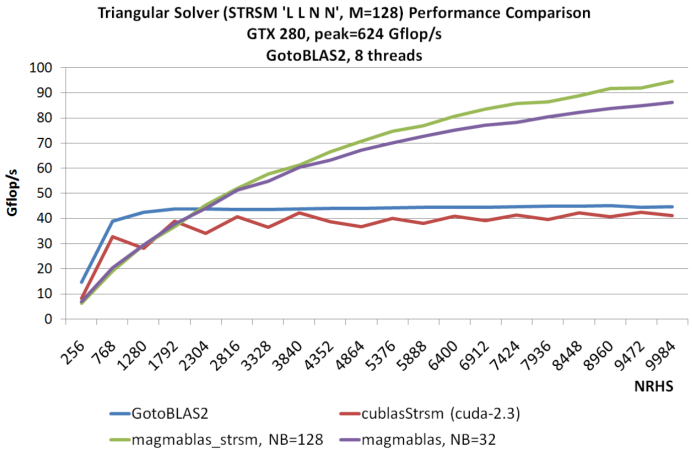

Fig. 18. Triangular solver performance on various hardware with different software implementations When $M < N$.

Lastly, we *empty* the TM update kernels, and count just the kernel launch time with regard to different NB. For each NB, kernels are called recursively, so we need the accumulated time of for every NB. Figure 16 shows this result. The kernel launch time does increases exponentially along with NB. But if this portion of time is removed, the leftover time (which is the net time spent in computation and communication, as shown by the red line in figure 16) still maintains the exponential increase trend. Also, since the total percentage of the kernel launch overhead over the total time actually decreases when NB increases, we conclude that this overhead does contribute to the time of diag_strtri(), but to a very minimal extent.

In conclusion, by decomposing the issue into several possible reasons and analyzing them one by one, it appears that the excessive growth of the time of diag_strtri() as NB increases is directly related to the complexity of the TM update, which is $O(n^2)$, and other factors only marginally affect the performance. Therefore, no further optimization for diag_strtri() would be able to make a significant change in run time.

VIII. Experimental Results

In a lot of cases cublasStrsm from CUBLAS is slower than the implementation on multicore CPUs and this forces TRSM to be done on CPU, which leads to extra memory copies. In our experiment, we compare the performance of our tuned TRSM, namely magmablas_strsm() with both multicore CPU implementation and cublasStrsm. The goal is to show that with help of CUDA profiler and TAU, our TRSM has been tuned to the performance where TRSM on CPU as an alternative is no long necessary.

The experiment environment was as follows:

- CPU: Intel Xeon E5410 (8 cores) @ 2.33GHz
- GPU: NVIDIA GeForce GTX 280 (240 cores) @ 1.3GHz
- BLAS: GotoBLAS2
- CUDA-2.3

And in our implementation we use single precision floating-point arithmetic [1].

For solving a system of linear equations of the form $Lx = b$ with a triangular matrix L, the size of L is M, and the Number of Right Hand Side (NRHS) is N. We show results for two different cases:

1) M is large but N is small, as found in LAPACK solver routines, like SGETRS, SGEQRS, etc.
2) M is small, but N is large, as found in LAPACK factorization routines, SGETRF, SPOTRF, etc.

Figure 17 shows the result of case 1. The two curves at the bottom show that cublasStrsm has similar performance to that of GotoBLAS2 and even worse if the time of device memory transferring is considered. By contrast, when using diagonal block inversion algorithm we see up to 10-fold performance increase for most of the tests. With the help of TAU and CUDA visual profiler to tune the multiple block size algorithm, up to 15-fold gain is obtained.

Figure 18 is the result of case 2. $M = 128$ is a common block size for some of the factorization routines in our MAGMA project [2]. Here, cublasStrsm is slower than GotoBLAS2 from the very beginning. Both of our versions have 100% improvement asymptotically to both cublasStrsm and GotoBLAS2. Our tuning for NB=128 gives it another 10% increase. This improvement is not as good as in case 1 because with $M = 128$, the GEMM that produces the solution only gets to work with small matrix sizes, which based on our profiling result leads to low performance and also the diagonal inversion is also performed on small matrices and the improvement becomes negligible. Also, note that with smaller NRHS, GotoBLAS2 is better. In the future overhead to transfer the data between the GPU and CPU memory will be evaluated, and the resulting algorithm could automatically switch between GotoBLAS2 and magmablas_strsm depending on the input matrices' shape to achieve the best performance.

In conclusion, the proposed TRSM algorithm with variable block size has shown good performance and with the help of CUDA visual profiler and TAU we were able to understand the roles that different parts of the solver play and accordingly

give further performance improvement.

IX. Related Work

Triangular system solvers are important for both factorizing system of equation and the subsequent back- and forward-solves. Unfortunately, the performance of cublasStrsm() available from NVIDIA in CUBLAS suffers from inefficiencies in the context of triangular solver and thus is not suitable for practical use. Consequently, there have been some efforts trying to improve the routine's performance. Explicit matrix inversion of diagonal submatrices that turns triangular solve into a matrix-matrix multiplication has been done in the context of developing factorization routines for GPUs [22], [20]: much better performance has been reported. The recursive algorithm that we used was proposed for triangular matrix inversion [18].

CUDA Profiler is used predominantly for measuring and profiling GPU performance but as we show, this has its shortcomings [11] – mostly related to the fact that only low-level information is available to the user. Higher level information can be obtained with TAU: since version 2.18.1 it can interface with the runtime accelerator library from PGI [12]. This allows to extract performance information associated with kernels that execute on the GPUs. Also, the development group of TAU continues work on a special module for CUDA called *TAUcuda* [14]. It is designed to measure the performance of GPU computations programmed using CUDA and integrate this information with application performance data captured with the TAU Performance System.

NVIDIA released a beta version of the Nexus add-on to Microsoft Visual Studio for the Windows Platform. This add-on is designed specifically to support CUDA C, OpenCL, and DirectCompute applications [16]. It is capable of capturing performance events and information across both CPU and GPU, and presenting the information to the developer on a single timeline. This makes it convenient for developers to see how their application behaves and performs as a whole across all hardware subsystems.

References

[1] ANSI/IEEE Standard 754-1985. Standard for binary floating point arithmetic. Technical report, Institute of Electrical and Electronics Engineers, 1985.

[2] Emmanuel Agullo, James Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series*, 180, 2009.

[3] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[4] E. Anderson, J. Dongarra, and S. Ostrouchov. Installation guide for LAPACK. Computer Science Dept. Technical Report CS-92-151, University of Tennessee, Knoxville, TN, March 1992. (Also LAPACK Working Note #41).

[5] Jack Dongarra and Joshua Hoffman. PLASMA TAU guide, 2010. Available electronically at http://icl.cs.utk.edu/projectsfiles/plasma/pdf/plasma_tau.pdf.

[6] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990.

[7] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:18–28, March 1990.

[8] Jack J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, March 1988.

[9] Jack J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, March 1988.

[10] J. Du Croz and N. J. Higham. Stability of methods for matrix inversion. *IMA J. Num. Anal.*, Jan 1992. (LAPACK Working Note #27).

[11] Rob Farber. Cuda, supercomputing for the masses: Part 6. *Dr. Dobb's Journal*, July 25 2008. Available at http://www.ddj.com/architect/209601096.

[12] The Portland Group. Pgi fortran & c accelerator programming model. Technical report, The Portland Group, July 2009. v1.0.

[13] B. Kågström, P. Ling, and Charles Van Loan. Portable high performance GEMM-based Level 3 BLAS. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 339–346, Philadelphia, 1993.

[14] Allen D. Mallony, Sameer Shende, Shangkar Mayanglambam, Scott Biersdorff, and Wyatt Spear. Performance measurement and analysis of heterogeneous parallel systems: Tasks and GPU accelerators. In *CSCaDS Summer Workshop 2009*, July-August 2009.

[15] NVIDIA. *NVIDIA CUDA™ Programming Guide Version 2.3.1*. NVIDIA, August 26 2009.

[16] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi v1.1. Technical report, NVIDIA Corporation, 2009.

[17] NVIDIA. Visual profiler user guide. May 2010. Available at http://developer.nvidia.com/object/cuda_2_3_downloads.html.

[18] Florian Ries, Tommaso De Marco, Matteo Zivieri, and Roberto Guerrieri. Triangular matrix inversion on graphics processing unit. In *Supercomputing SC09*, Portland, Oregon, USA, November 14-20 2009.

[19] S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[20] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. Technical report, LAPACK Working Note 210, October 2008.

[21] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing 08*. IEEE, 2008.

[22] Vasily Volkov and James W. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department,University of California, Berkeley, May 13 2008. LAPACK Working Note 202.

[23] Michael Wolfe. Compilers and more: Optimizing GPU kernels. *HPCwire*, October 30 2008.

[24] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proceedings of SC2000: High-Performance Networking and Computing*, November 2000.